Highly-Available, Fault-Tolerant, Parallel Dataflows

Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer

IRB-TR-04-002

February, 2004

Intel Research Berkeley

Highly-Available, Fault-Tolerant, Parallel Dataflows

Mehul A. Shah
U.C. Berkeley
mashah@cs.berkeley.edu

Joseph M. Hellerstein
U.C. Berkeley
Intel Research, Berkeley
jmh@cs.berkeley.edu

Eric Brewer
U.C. Berkeley
brewer@cs.berkekey.edu

ABSTRACT

We present a technique that masks failures in a cluster to provide high availability and fault tolerance for long-running, parallelized dataflows. We can use these dataflows to implement a variety of CQ applications that require high-throughput, 24x7 operation. Examples include network monitoring tasks, phone call processing, click-stream processing, and online financial analysis. Our main contribution is a scheme that carefully integrates traditional query processing techniques for partitioned parallelism with the process-pair approach for high availability. Our approach replicates and coordinates dataflow operators on a per-partition basis. Upon failure, our technique provides quick failover, and recovers the dataflow on-the-fly without stalling the progress of the ongoing computation and deteriorating result quality. This delicate integration also results in a reduced mean time to recovery (MTTR) over the straightforward application of the process-pair technique on a per-dataflow basis, thereby improving overall mean time to failure (MTTF). Our techniques are encapsulated in a reusable dataflow operator called Flux , an extension of the Exchange. Flux is interposed at the communication points between existing operators in a partitioned parallel dataflow. Encapsulating the fault-tolerance logic into Flux minimizes modifications needed to existing operator code and relieves the operator writer of the burden of repeatedly implementing and verifying critical recovery logic. We present experiments illustrating these features using an implementation of Flux in the TelegraphCQ code base [8].

1. INTRODUCTION

There are a number of continuous query (CQ) or stream processing applications that require high-throughput, 24x7 operation. One important class of these applications includes critical, online monitoring tasks. For example, to detect attacks on hosts or websites, individual flows are reconstructed from network packets and the flows' contents are inspected [25, 18]. Another example is processing a stream of call detail records for telecommunication network management [17, 3]. Phone billing systems perform various data management operations using call records to charge or even route phone calls. Websites may analyze click streams in real-time for user targeted marketing or site-use violations [31]. Other applications include financial quote analysis for real-time arbitrage opportunities, monitoring manufacturing processes, and instant messaging infrastructure. Recent research suggests that we can implement these applications using a more general CQ infrastructure that produces continuously processing dataflows [9, 7, 24, 8]. A CQ dataflow is a DAG in which vertices represent operators that process incoming data and edges denote the direction in which data is passed.

For such critical, long-running dataflow applications, scalability, high availability, and fault tolerance are the primary concerns. A viable approach for scaling these dataflows is to parallelize them across a shared-nothing cluster of workstations. Clusters are a cost-effective, highly-scalable platform [10, 29, 2] with the potential to yield fast response times and permit processing of high-throughput input streams. However, since these dataflows run for an indefinite period, they are bound to experience machine faults on this platform, either due to unexpected failures or planned reboots. A single failure can defeat the purpose of the dataflow application altogether. Long interruptions in computing results, dropping incoming or in-flight data, or losing accumulated operator state are all unacceptable behaviors. In an intrusion detection scenario, for example, any one of these problems could lead to hosts being compromised. For streaming financial applications, even short interruptions can lead to missed opportunities and quantifiable revenue loss. Hence, in addition to scalability, such applications need a fault-tolerance and recovery mechanism that is fast, automatic, and non-disruptive.

To address this problem, we present a mechanism called Flux that masks machine failures to provide high availability and fault tolerance for dataflows parallelized across a cluster. Flux has following the salient features.

Flux interleaves traditional query processing techniques for partitioned parallelism with the process-pair [15] approach of replicated computation. Typically, a database query plan is parallelized by partitioning its constituent dataflow operators and their state across the machines in a cluster, a technique known as partitioned parallelism [23]. The main contribution of Flux is a technique for correctly coordinating replicas of individual operator partitions within a larger parallel dataflow. In this paper, we address the challenges of avoiding deadlock and maintaining exactly-once, in-order delivery of the input to these replicas during failure and recovery. As a result of this delicate integration, our technique yields a lower mean time to recovery (MTTR) than the straight-forward approach of simply coordinating replicas of an entire dataflow, thereby improving the reliability, or mean time to failure (MTTF), of the system.

In addition to quick failover (a direct benefit of replicated computation) Flux also provides automatic on-the-fly recovery that limits its disruptions to ongoing dataflow processing. As part of recovery, Flux copies lost state from remaining replicas onto a spare machine to restore redundancy for the failed portion of the dataflow. Flux restores lost in-flight data and accumulated operator state without de-

grading result quality or stalling the ongoing computation. The dataflow processing continues for unaffected partitions. These features provide the high availability necessary for critical applications that not only cannot tolerate inaccurate results and data loss but also have low latency requirements. For restoring lost state, Flux leverages an API for extracting and installing the state of operator partitions that must be implemented by the operator writer.

Inspired by Exchange [13], Flux encapsulates the coordination logic for failover and recovery into an opaque operator used to compose parallelized dataflow operators. Flux is a generalization of the Exchange [13] operator, the communication abstraction used to compose partitioned-parallel query plans. This technique of encapsulating the fault-tolerance logic, allows Flux to be reused for a wide variety of database operators including stateful ones. This design relieves the burden on the operator writer of repeatedly implementing and verifying critical fault-tolerance logic. By inserting Flux into the communication points within a dataflow, an application developer can make the entire dataflow robust.

In this paper, we describe the Flux design and recovery protocol in detail and illustrate its features. In Section 2, we start by stating our assumptions about the processing model, the platform and fault model, and outline our basic approach for fault-tolerance. Next, in Section 3, we describe how to apply the naive process-pair approach to single-site dataflows. Section 4 describes the necessary modifications to Exchange for operating in a stream processing environment, and the problems with a naive application of the process-pairs technique to partitioned parallel dataflows. In Section 5, we build on the protocols and techniques developed in the previous sections to describe Flux 's design and its recovery protocol. We also demonstrate the benefits of Flux with experiments using an implementation of Flux in the TelegraphCQ code base. Section 6 summarizes the related work. Section 7 summarizes and presents future work.

2. ENVIRONMENT AND SETUP

In this section, we outline the model for dataflow processing and our basic approach for achieving fault-tolerance and consistency. We also describe the underlying platform, the faults we guard against, and services upon which we rely. Finally, we present a motivating example used throughout the paper.

2.1 Processing Model

A CQ dataflow is a generalization of a pipelined query plan. It is a DAG in which the nodes represent "non-blocking" operators, and the edges represent the direction in which data is processed. A *stage* in the dataflow is a producer-consumer operator pair and the mechanisms that connect them together.

A CQ dataflow is typically a portion of a larger end-to-end dataflow. As an example, consider a packet sprayer that feeds data to a CQ dataflow that monitors for intrusions and whose output is spooled to an administrator's console or log. In this example, the larger dataflow includes the users or machines generating packets and the administrators that receive notifications. In this paper, we only concentrate on the availability of a CQ dataflow and not the end-to-end dataflow nor the entry and exit points. We focus on CQ dataflows that receive input data from a single entry point and return data to clients through a single exit, thereby allowing us to impose a total order on the input and output data. Also, for clarity, we only discuss linear dataflows, although our techniques extend to arbitrary dataflows. For the parallel, cluster-based setting, we assume a partitioned dataflow model where CQ operators are declustered across multiple nodes, and multiple operators are connected in a pipeline [23].

CQ operators process over infinite streams of data that arrive at their inputs. Operators export the Fjord interface [22], `init()` and `processNext()` which are similar to the traditional iterator interfaces. When the operator is invoked via the `processNext()` call, it performs some processing and returns tuple if available for output. However, unlike `getNext()` of the traditional iterator interface, `processNext` is non-blocking. The `processNext` method need not return data, but should make an effort to do a small amount of work and quickly return control to the caller.

Operators communicate via queues that buffer intermediate results. The operators can be invoked in a bottom-up, pushed-based fashion as input arrives from the source or in a top-down, pull-based fashion starting from the output, or some combination as determined by a scheduler [22, 7]. Most CQ operators have one or more inputs and a single output. Certain "gluing" operators like Exchange or Flux have multiple outputs, and these are used to connect together other CQ operators.

2.2 Platform Assumptions

In this section, we describe our platform, the types of faults that we handle, and cluster-based services we rely upon external to our mechanism. For this work, we assume a shared-nothing parallel computing architecture in which each processing node (or site) has a private CPU, memory, and disk, and is connected to all other nodes via a high-bandwidth, low-latency network. We assume the network layer provides a reliable, in-order point-to-point message delivery protocol, e.g. TCP. Thus, a connection between two operators is modeled as two separate uni-directional FIFO queues, whose contents are lost only when either endpoint fails.

We ignore recurrent deterministic bugs and only consider hardware failures or faults due to "heisenbugs" in the underlying platform. These faults in the underlying runtime system and operating system are caused by unusual timings and data races that arise rarely and are often missed during the quality testing process. When these faults occur, we assume the faulty machine or process is *fail-stop*: the error is immediately detected and the process stops functioning without sending spurious output. Schneider [27] and Gray and Reuter [15] show how to build fail-stop processes. Moreover, since we aim to provide both consistency and availability, we cannot guard against arbitrary network partitions[12].

We rely on a cluster service that allows us to maintain a consistent global view of the dataflow structure and active nodes in the cluster [32]. Such a service usually employs a highly-available, atomic consensus protocol that can be used to maintain a modest amount of highly-available global metadata [20]. We call it the *controller*. Note, the controller is not a single point of failure, but a uniform view of highly-available service. We rely on it to update group membership to reflect active, dead, and standby nodes, setup and teardown the dataflow, and perform consistent updates to the dataflow structure. It also has the responsibility of making decisions about changes in the dataflow structure during failures, e.g. assigning operators to standby nodes after a failure. Most importantly, our recovery techniques rely on this service to detect and report machine failures and machine availability. We assume that such messages from the controller arrive at all cluster nodes in the same order.

We do not, however, rely on the service to maintain or recover any information about the in-flight data or internal state of the dataflow operators. The initial state of operators must be made persistent by external means, and we assume it is always available for recovery. Our techniques also do not rely on any stable storage for making the transient dataflow state fault-tolerant. When nodes fail and re-enter the system, they are stateless.

2.3 Basic Fault-Tolerance Approach

Our goal is to make the in-flight data and transient state of dataflow

operators fault-tolerant and highly available. In-flight data are all tuples in the system from acknowledged input from the source to unacknowledged output to the client. This in-flight data includes intermediate output generated from operators within the dataflow that may be in local queues, network buffers, or within the network itself.

Our approach for fault-tolerance and high-availability is variant of process-pair approach [15] or the more general technique of replicated state machines [19]. The basic idea is to have redundant copies of each piece of computation and properly coordinate the pieces upon failure and during recovery. Throughout this paper, our discussion will be restricted to techniques for coordinating two replicas; thus, we will be able to tolerate a single failure. While further degrees of replication are possible with our technique, for most practical purposes, the reliability achieved with pairs, which tolerate a single failure between recovery points, is sufficient [15].

We model each CQ operator as a deterministic state machine. Through its execution, an operator moves through a sequence of states. This sequence is determined solely by the initial state of the operator and the sequence of tuples the operator has consumed. In this definition, we only consider the data on inputs and outputs visible from the dataflow. We assume that replicating the input data in the same order to distinct instances of an operator will produce identical output in identical order, and both will be in the same final state. We call this property of an operator *sequence-preserving*. Most CQ operators fall within this category, for example, windowed symmetric hash join or windowed group-by aggregates.

Our techniques ensure that replicas are kept consistent by properly replicating their input stream during normal processing, upon failure, and after recovery. With the dataflow model, input ordering is fairly straightforward to maintain, given an in-order communication protocol. However, consistency is difficult to maintain during a failure and after recovery because connections can lose in-flight data and operators may not be perfectly synchronized. Thus, our techniques maintain the following two invariants in order to achieve this goal.

1. Loss-free: no tuples in the input stream sequence are lost.

2. Dup-free: no tuples in the input stream sequence are duplicated.

Some operators exhibit external behavior that depends on inputs outside of the scope of the dataflow. For example, the output order of XJoin [30] results depend upon the prevailing memory pressures. Because we cannot capture and mimic all external inputs, such operators behave non-deterministically from the perspective of the dataflow. We will see in Section 3.5 that with slight modifications, our techniques for replicated single-site dataflows can accommodate a subset of non-deterministic operators that obey the *set preserving* property. That is, given the same *set* of input tuples, the operator will produce the same *set* of output tuples.

2.4 Motivating Example

As a motivating example, we consider a dataflow that may be used for analyzing click stream sessions in real time. Imagine a website that wants to track the maximum and average user session duration over time, on per-user basis. Such information may be used, for example, to detect anomalous behavior or generate targeted advertisements. The linear dataflow in Figure 1 performs this computation; we will use this as our example throughout the paper.

The data source, the first operator, provides a raw click stream. To determine session duration, the user session must be reconstructed from a click stream. Each click stream tuple has the schema, `(sess-id, user-id, data, ts)`, where `sess-id` (session id) is the key, `ts` is a timestamp, and the `data` field contains activity information or signals a start or end of session. Thus, the second operator is a streaming group-by aggregation that outputs a `(sess-id, user-id, duration)`. The third operator is a group-by agg that maintains `(user-id, max(duration), avg(duration))` for each user and outputs a new tuple at a frequency dictated by an administrator. The final operator is an output operator.

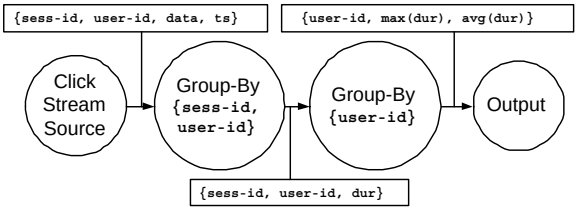

Figure 1: *Example Dataflow*

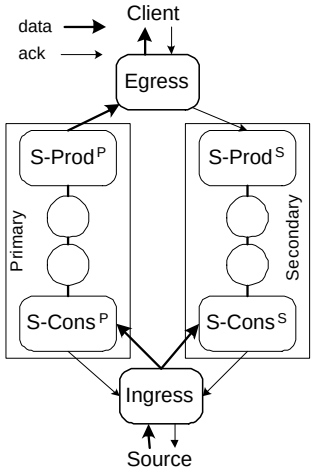

Figure 2: Dataflow Pairs Normal Processing

3. SINGLE-SITE DATAFLOW

The process-pair approach is a common primary-backup fault-tolerance technique that provides quick fail-over and thus high-availability [15]. In this section, we describe how to apply this technique to build a replicated, single-site CQ dataflow. We extend the technique to parallel dataflows in the next section.

For a given CQ dataflow, we introduce additional operators that coordinate replicas of the dataflow during normal processing and automatically perform recovery on upon machine failure. We refer to these operators as single-site fault-tolerance (SSFT) operators. These operators encapsulate this logic so modifications to existing operators are unnecessary. Recovery has two phases: *take-over* and *catch-up*. Immediately after a failure is detected, take-over ensues. During take-over, we adjust the routing of data within the dataflow to allow the remaining replica to continue processing and generate and deliver results. After take-over, the dataflow is vulnerable: one additional failure would cause the dataflow to stop. Once a standby machine is available, the catch-up phase creates a new replica of the dataflow, making it once again fault-tolerant.

3.1 Dataflow Pairs

In this section, we describe the normal-case processing and coordination necessary between replicated dataflows to guard against failures.

3.1.1 Components of a Single-Site Dataflow

A single-site CQ dataflow is composed of a pipeline of operators, and this pipeline is typically in a single thread of control. We call any such locally connected sequence of operators a *dataflow segment*, as shown in the left portion of Figure 2. Communication to non-local machines occurs only at the top and the bottom of the dataflow segment. When we refer to the top, we mean the end that delivers output and the bottom is the end that receives input. Likewise, we use above to indicate closer to the output, and vice-versa for below.

In top-down execution, the topmost operator invokes the operators below it recursively through the `processNext()` method, and returns the results through an *egress* operator that forwards results to the client. At the bottom are operators that receive data from an *ingress* operator. Ingress handles the interface to the network source. The circles are operators in the dataflow; in our running example they are group-by operators.

In this paper, we place the ingress and egress operators on separate machines and assume they are always available. These operators are proxies for the input and output of the dataflow and may have customized interfaces to the external world. Existing techniques, e.g. process pairs, may be used to make these operators highly-available, but we do not discuss these further. Our focus is on the availability of the dataflow that processes the incoming data and its interaction with these proxy operators. Thus, in the remaining discussion, we detail ingress and egress alongside the other SSFT operators we introduce.

3.1.2 Normal Case Protocol

The ingress operator is responsible for incorporating the network input into data structures accessible to the dataflow execution engine, i.e. tuples. Once the ingress operator receives an input and has incorporated it into the dataflow, it sends an acknowledgment to the source indicating that the input is stable and the source can discard it. For push-based sources that do not process acks, the operator does its best to incorporate the incoming data.

The egress operator does the reverse. It sends the output to the client and when an acknowledgment is received from the client, the egress operator can discard the output. If the client does not send acknowledgements, result delivery is also just best-effort.

To make a single-site dataflow fault-tolerant and highly-available, we replicate the entire dataflow on a separate machine and forward input to both to loosely synchronize the processing. Henceforth, we use the superscript P to denote the primary copy of an object, and the superscript S to denote the secondary (see Figure 2).

In the normal case, the egress and ingress operators poll for input, forward it to the appropriate destinations, and send acknowledgements (abbreviates as acks). We introduce additional two routing operators called S-Prod and S-Cons at the top (producing) and at the bottom (consuming) end of the dataflow segment respectively. We assume each input tuple is assigned a monotonically increasing input sequence number (ISN) from the source or ingress operator. The ingress operator buffers and forwards each stable input tuple to both S-Cons^P and S-Cons^S which upon receipt send acks. The S-Cons operators do not forward an incoming tuple to operators above unless an ack for that tuple has been sent to the ingress. Acknowledgements, unless otherwise noted, are just the sequence numbers assigned to tuples. When ingress receives an ack from both replicas for a specific ISN, it drops the corresponding input from its internal buffer. Likewise, both S-Prod operators assign an output sequence number (OSN) to each output and store the output in an internal buffer. Only S-Prod^P forwards the output to the egress operator after which it immediately drops the tuple. Egress sends acks to S-Prod^S for every input received. Once an ack is sent, egress can forward the input to the client. Once S-Prod^S has received the ack from egress, it discards the output tuple with that OSN from its buffer. Note the asymmetry in this forwarding protocol. We will see in Section 5, it is exactly half of the Flux protocol, which is symmetric.

Clearly, this protocol ensures that both dataflow replicas receive the input in the same order, but it also is sufficient for ensuring our loss-free and dup-free invariants. It maintains these invariants during and after failures by managing the pending output tuples and acknowledgements in the internal buffers of the ingress and S-Prod operators. This protocol ensures that every input tuple is replicated on at least two different nodes at all times and similarly for each output tuple. Based on this property, we will see that the in-flight data is fault-tolerant. The buffer also serves as a duplicate filter to avoid re-sending redundant output during and after recovery. First, we describe the interface these buffers support, and how the operators use them to perform proper accounting for in-flight data. Then, we describe how these buffers are used during recovery.

3.2 Using Buffers

The buffer in the ingress operator stores sequence numbers and associated with those sequence numbers are tuples and markings. If sequence number does not have a tuple associated with it, we call it a *dangling* sequence number. The markings indicate the places from which the sequence number was received. A marking can be any combination of `PROD`, `PRIM`, and `SEC`. The `PROD` mark indicates it is from a tuple produced from below. The `PRIM` and `SEC` markings indicate it was from an ack received from the primary and secondary destinations, respectively. The buffer also maintains two cursors: one each for the primary and secondary destinations. The cursors point to the first undelivered sequence number to each destination. The buffer in S-Prod is the same, except there is only one destination, so only one cursor exists and only two markings are allowed, `PROD`, `PRIM`. The buffer supports the following methods: `peek(dest)`, `advance(dest)`, `put(Tuple,SN,del)`, `ack(SN,dest,del)`, `ack_all(dest,del)`, `reset(dest)`.

The `peek()` method returns the first undelivered tuple for the destination and `advance()` moves the cursor for that destination to the next undelivered tuple. The `put()` method inserts a sequence number, SN, into the buffer if the SN does not yet exist. It then associates a tuple with that sequence number, and marks that tuple as produced. The `ack()` method marks the sequence number as acknowledged by the given destination. The `del` parameter for both `put()` and `ack()` indicates when to purge a sequence number and its associated tuple from the buffer. This parameter specifies markings for the SN that must exist in order to remove its entry. The `reset()` and `ack_all()` methods are used during take-over and catch-up, so we defer their description to Section 3.3.

Our operators that produce or forward data use this buffer and implement the abstract specification in Figure 3. The specification contains state variables and actions that can modify the state or produce output. State variables can be scalar or set-valued. The values they can hold are declared within braces, and for set-valued types, these values are separated by bars. Each row in the specification is an action. Each action has a guard, a predicate that must be true, to enable the action. Each enabled action causes a state change. State change commands surrounded by braces imply the sequence of commands is atomic. If multiple actions are enabled, any one of their state changes may occur. Indented predicates are shorthand for nested conditions; the higher level predicate must also be true. If the nested condition is also true, their corresponding commands are appended to the higher level commands. Actions may also have external actions as pre-conditions (see Figure 4). While the abstract

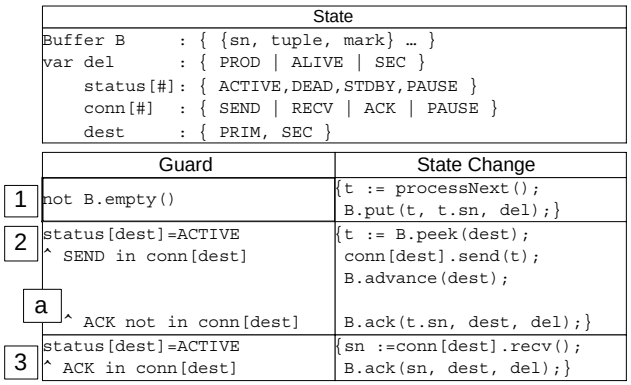

Figure 3: Abstract Producer Specification - Normal Case

prodcuer description only specifies actions related to the output side of the operators, it suffices to illustrate how the buffer is used.

The specification in Figure 3 indicates that any one of three actions may occur in a data forwarding operator like ingress or S-Prod . If there is room in the buffer, the operator gets the next incoming tuple, determines the sequence number, and inserts it using `put()` (action (1)). If a destination is alive and its connection indicates that it should send, then it removes the first undelivered tuple, sends it, and advances the cursor (action (2)). If the connection also is not acking, the tuple's sequence number is immediately acked after sending (action (2a)). If the connection indicates that acks should be processed, the next acked sequence number is retrieved and is stored in the buffer (action (3)).

By setting the variables appropriately, we obtain the behavior of our SSFT producing operators. For the ingress operator, the `dest` variable ranges over both primary and secondary connections. For both connections, the `conn` variable is {`SEND,ACK`} to indicate both sending and acking should occur. The `del` variable is {`PROD,PRIM,SEC`} to indicate that all three markings are necessary for eviction. For S-Prod^P , `conn[PRIM]= SEND` and for S-Prod^S , `conn[PRIM]=ACK`, i.e. data is not forwarded. For the S-Prod operators, `dest=PRIM` always, and `del` is set to {`PROD, PRIM`}. Thus, for S-Prod^P , entries in the buffer are deleted only after having been produced and sent. For S-Prod^S , entries are deleted after having been produced and acked. Note, for S-Prod^S , acks from egress may be inserted in the buffer even before their associated tuples are produced.

The size of the ingress buffer limits the drift of the two replicas; the shorter the buffer the less slack there is between the two. Since we assume underlying in-order message delivery, the buffers can be implemented as simple queues, and the acks can be sent periodically by S-Cons or egress to indicate the latest tuple processed. In this case, `ack()` acknowledges every tuple with sequence number $\leq$ SN. This scheme allows us to amortize the overhead of round-trip latencies. With this optimization, the take-over and catch-up protocols must change somewhat. We omit these details due to space constraints.

The mean-time-to-failure (MTTF) analysis for the dataflow pair during normal processing is the same as that of process pairs, assuming independent failures. The system's MTTF is $(\text{MTTF}_s)^2/\text{MTTR}_s$ where MTTR_s is the time to recover a single machine, MTTF_s is the MTTF for a single machine, and $\text{MTTR}_s \ll \text{MTTF}_s$ [15]. In our case, MTTR_s consists of both the take-over and catch-up phase. The latter can only occur if a standby is available.

3.3 Take-Over

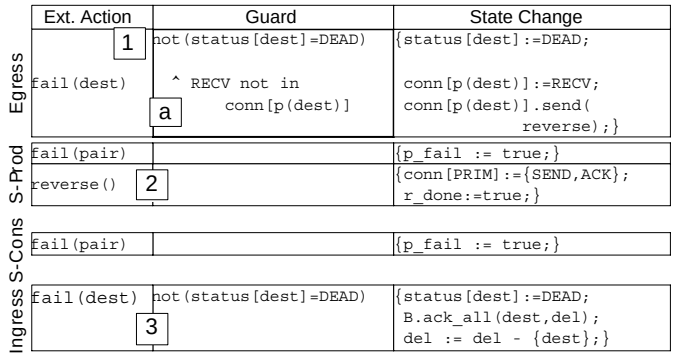

Figure 4: Take-Over Specification

In this section, we describe the actions involved in the take-over protocol for our routing operators. We being by discussing how controller messages are handled and what additional state the routing operators must maintain. We then describe the protocol and show how it maintains our two invariants after a failure.

3.3.1 Controller Messages and Operator Modes

As mentioned previously, we rely on the controller to detect and report failures. Typically, such a service will detect failures via timeout of periodic heartbeats, and inform all active nodes of the failed node through a distributed consensus protocol. We model this as an external action `fail()` that is invoked on our routing operators. The controller also sends availability `avail()` messages used during catch-up (see Section 3.4). Since the controller can send multiple messages, we assume that all enabled actions for a controller message complete before actions for the next message can begin.

While the controller ensures that the fail message arrives at all machines in same order within all controller commands, it makes no guarantee as to when that message will arrive at any particular node with respect to data routed within a dataflow. For example, the message may arrive at the ingress operator before some tuple t has been forwarded and arrive at a S-Cons operator much after the tuple t has been received. Thus, our SSFT operators must coordinate using messages within the dataflow to perform take-over.

Moreover, our SSFT operators must maintain the status of the operator on the other end of all outgoing and incoming connections. Each operator can be in four distinct modes: `ACTIVE`, `DEAD`, `STDBY`, `PAUSE`. In the `ACTIVE` mode, the operator is alive and processing. When it is dead, it is no longer part of the dataflow. We discuss the `STDBY` and `PAUSE` modes in the discussion of catch-up in Section 3.4.

3.3.2 Re-routing Upon Failure

Upon failure, we need to make two adjustments to the routing done by our SSFT operators to ensure the remaining dataflow segment continues computing and delivering results. First, in the ingress operator, we must adjust the buffer to no longer account for the failed replica. Second, if the primary dataflow segment fails, the secondary S-Prod must forward data to egress. The actions enabled upon a failure and during take-over for our SSFT operators are shown in Figure 4. We describe these top-down in our dataflow.

When a failure message arrives at the egress operator, first it simply adjusts the connection status state to indicate that the destination is dead (action (1)). If the connection to the remaining replica, indexed by `p(dest)`, is receiving results, nothing else happens. Otherwise, the egress operator marks that connection for receiving tuples. Egress also sends a `reverse` message to the replica (action

(1a)). Hence, if the primary dataflow segment fails, egress begins processing results from the secondary. The reverse message is sent along the connection to S-Prod that was used for acks. This message is an indication to S-Prod to begin forwarding tuples instead of processing acks.

When a reverse marker arrives at the S-Prod , it simply adjusts the connection state to begin sending to its only connection (action (2)). Using the buffer ensures that no results are lost or duplicated. Once the reverse message arrives, there are no outstanding acks from egress, given our assumption about in-order delivery along connections. At this point, there can be two types of entries in the buffer: unacknowledged tuples, and dangling sequence numbers. The dangling sequence numbers are from received acks that have tuples yet to be produced. Clearly no tuples are lost because all lost in-flight tuples from the primary either remain unacknowledged in the buffer or will eventually be produced, assuming the ingress stage is fault-tolerant. Moreover, no tuples with acknowledged sequence numbers will be resent because the buffer acts as a duplicate filter. When a tuple is produced it is inserted using the `put()` method, and if a dangling sequence number exists for it, it is immediately removed.

When S-Prod starts sending, the buffer ensures the primary's cursor points to the first undelivered tuple in the buffer, i.e. the first unacknowledged tuple. As described in the next section, the S-Prod and S-Cons operators maintain additional state, `p_fail`, used to indicate the completion of the take-over phase.

Upon receiving the failure message, the ingress operator first sets the connection to dead (action (3)). Then, using the `ack_all()` method, it marks all tuples destined for the dead connection as acknowledged. Like `ack()`, `ack_all()` will remove tuples containing all the marks in `del`. Finally, ingress adjusts the `del` to only consider the produced marker, `PROD`, and one of `PRIM` or `SEC` for the remaining connection, when evicting tuples. No tuples are lost or duplicated because for the remaining connection, the ingress operator is performing exactly the same actions. Moreover, the buffer does not fill indefinitely after take-over because `del` is adjusted.

After take-over is complete, the dataflow continues to compute and deliver results. However, it is vulnerable to an additional failure.

3.4 Catch-Up

To make the dataflow fault-tolerant again, we need another non-faulty machine that has the dataflow and operators initialized in `STANDBY` mode. We assume that the controller arranges such a machine after failure and issues an `avail(mach)` message to indicate its availability. We also assume that it initializes the standby machine with operators in their initial state and has the connections properly setup for the routing operators.

Once a standby dataflow segment is available, the catch-up phase commences. We first give an overview of this phase. Initially, the S-Cons operator quiesces the dataflow segment and begins state movement. Then, state movement is accomplished through a StateMover that is local to the machine, but in a separate thread outside the dataflow. The StateMover has a connection to the StateMovers on all machines in the cluster. S-Cons leverages specific API for extracting and installing the state of the dataflow operators. Once state movement is complete, both active and standby S-Prod and S-Cons operators send synchronization messages to the egress and ingress operators respectively. The messages mark the point, within the output or input streams of the dataflow segment, at which the two new replicas are consistent. Once the ingress and egress operators receive these messages from the active and standby, the incoming and outgoing connections are restarted and catch-up is complete. Figure 5 shows the specification for the catch-up phase. We describe these bottom-up, as they occur.

The S-Cons operator initiates the catch-up phase when it recognizes that a standby dataflow segment is ready via an `avail()` message from the coordinator. The S-Cons operator both on the active and the standby relies on a local, but external StateMover process for transferring state. The S-Cons that sends its state is in `ACTIVE` mode, and the S-Cons on the standby machine is in `STDBY` mode. The active S-Cons operator checks with its S-Prod above to determine if take-over has completed (action (1)). If so, it signals the StateMover process to begin state movement. These StateMovers on the standby and active machines communicate, and, when ready for transferring state, they both signal their respective S-Cons with `sm_ready` (action (2)). Next, S-Cons increments its version number and pauses the incoming connection. It also calls `initTrans()` which is invoked on every operator in the dataflow segment to quiesce the operators for movement. Eventually the

	Ext. Action	Guard	State Change
Egress	`avail(dest)`		`{status[dest]:=STDBY; conn[dest] := {};}`
	`psync(dest,v)` 8	`status[dest]=ACTIVE ^ ver[p(dest)] = v`	`{ver[dest] := v; conn[p(dest)]:={ACK};}`
		`else`	`conn[dest]:={PAUSE};}`
		`status[dest]=STDBY ^ ver[p(dest)] = v`	`status[dest]:=ACTIVE;} conn[p(dest)]:= RECV; conn[dest] := ACK;}`
S-Prod	`tk_done()`	`p_fail ^ r_done`	`{return true;}`
	`sync(ver)` 7	`my_status=STDBY`	`{p_fail:=r_done:=false; send psync(ver); my_status:=ACTIVE; conn:={ACK};}`
	`initTrans()`	3	`{p_status:=my_status; my_status:=PAUSE;}`
	`endTrans()`		`{my_status:=p_status;}`
S-Cons	`avail(pair)` 1	`my_status=ACTIVE ^ p_fail ^ S-Prod.tk_done()`	`{send state move req}`
		`my_status=STDBY ^ p_fail`	`{send state move req}`
	`sm_ready()` 2		`{my_ver:=my_ver+1; conn:=PAUSE; initTrans(); enable move();}`
	`move` 4	`my_status:=ACTIVE` `my_status:=STDBY`	`{getState(),send data}` `{get data,installState()}`
	`sm_done()` 5		`{endTrans(),p_fail:=false; conn:= {RECV, ACK}; S-Prod.sync(my_ver); send csync(my_ver); my_status:=ACTIVE;}`
Ingress	`avail(dest)`		`status[dest]:=STDBY`
	`csync(dest, v)` 6	`status[dest]=ACTIVE ^ ver[p(dest)]=v`	`{ver[dest]:=v; del:=del+{p(dest)}; B.reset(dest); conn[p(dest)]:={SEND, ACK};}`
		`status[dest]=STDBY ^ ver[p(dest)]=v`	`{status[dest]:=ACTIVE; conn[dest]:={}; ver[dest]:=v; conn[dest]:={SEND, ACK};}`

Figure 5: Catch-Up Specification

call reaches the S-Prod above and pauses the operator. Then state movement begins.

At this point, there are two important items to note. First, S-Prod pauses to prevent additional state changes from occurring during movement (action (3)). Assume `not PAUSE` is a guard for every action; thus, the dataflow segment is completely stalled. Second, we have introduced version numbers for each dataflow segment. The version number tracks the number of state movement phases that have begun. Ingress and egress also maintain the version for the segments on the other of both connections. During state movement, the version number is copied so the active's and standby's version values match after movement. These matching version numbers are then used to properly match the synchronization messages that are sent after state movement completes.

In order to transfer the state of the intermediate operators, we require that operator developers implement two main methods: `getState()` and `installState()` (action (4)). These methods marshall and extract, and unmarshall and install the state of the operator. S-Cons calls these methods on every dataflow operator during state movement. For example, for the group-by from our example, `getState()` extracts the hash table entries that contain intermediate aggregation state and `installState()` does the reverse. Even S-Prod implements these methods to transfer its internal buffer. Similarly, `initTrans()` and `endTrans()` are needed to initiate and end transfers. These latter methods quiesce and restart the operator.

Once state transfer is finished, the `sm_done` action is enabled (action (5)). Next, both S-Cons operators restart their ancestors, restart their input connection for receiving and acking, enable `sync()` on their S-Prod above, and send a `csync` message to the ingress operator. Note, the `csync` messages are sent along the connection to the ingress for sending acks, flushing all acks to the ingress. At this point, both S-Cons are active, but catch-up is not quite complete.

The ingress operator completes catch-up after both `csync` messages arrive (action (6)). We are careful, however, not to restart the connection to the standby segment until the `csync` from the active dataflow segment arrives. The `ver` variable is used to track whether the `csync` was received. When it arrives, we are certain that all acks sent by the active dataflow segment prior to state transfer have been received; thus, only unacknowledged tuples remain in the buffer. Otherwise, we would duplicate previously consumed tuples to the new replica. These first of the unacked tuples in the buffer is where the output to the new replica segment must begin. The `reset()` method in the buffer starts the cursor for the new replica exactly at that point. Catch-up is complete for ingress when it activates the connection to the new replica and updates the `del` variable to indicate both acks are necessary for eviction.

Returning to the top of the dataflow segment, once S-Prod operators receive the `sync` from the S-Cons below, they emit a `psync()` with version numbers along the forward connection to egress (action (7)). After this message is sent, both S-Prod are active, and only catch-up for the egress remains to be discussed.

At the egress operator, we must ensure that it does not forward acks for in-flight data sent before movement began and that it does not miss sending acks for tuples sent after movement completed. If `psync` arrives from the standby first, we are careful not to activate the connection for acks (action (8)); otherwise, egress would forward acks for tuples sent before movement. If the `psync` arrives from the active first, we are careful to pause the connection until the second `psync` arrives. Otherwise, egress would miss sending acks for tuples received after state movement but before the standby `psync` arrives. The second `psync` causes both connections to be activated. The new replica is then sent acks, and catch-up is complete.

3.5 Conclusion - Dataflow Pairs

There are two properties of this protocol that we want to highlight. First, with some more modifications, we can make the catch-up protocol idempotent. That is, even if the standby fails during movement, the dataflow will continue to process correctly and attempt catch-up again. The changes are as follows. On a failure during movement, the S-Cons operator completes the protocol as if movement did finish. We must augment the version numbers with unique machine ids to ensure that spurious syncs from previous failed tries are disregarded. And when `fail()` arrives, the operators must update their state for the unsuccessful standby appropriately. Idempotency can be extremely useful. For example, imagine an administrator wants to migrate the dataflow to machine with an untested upgraded OS. With such a property, she can simply terminate a replica, bring up a standby with the new OS, and if the standby fails, revert back to the old OS.

Second, note the entire protocol only makes use of sequence numbers as unique identifiers; we never take advantage of their order, except in the optimizations for amortizing the cost of round-trip latencies. If we ignore these optimizations, we can accommodate dataflow segments that include non-deterministic but set-preserving operators, e.g. XJoin [30] and an Eddy [26]. However, instead of generating new OSNs at the output, we require that outputs have a unique key which can be used as sequence numbers in our protocol. We outline methods for maintaining such keys in Section 4.1.

4. PARALLEL DATAFLOW

Parallelizing a CQ dataflow across a cluster of workstations is a cost-effective method for scaling high-throughput applications implemented with CQ dataflows. For example, in our click stream scenario, one can imagine having thousands of simultaneous sessions and thousands of users. Moreover, the statistics (e.g. max) collected may range over some considerable window of history. To keep up with high-throughput sources and maintain low-latencies, the dataflow can be scaled by partitioning it across a cluster.

In a cluster-based system, a pipelined CQ dataflow is a collection of dataflow segments (one or more per-machine). Individual operators are parallelized by partitioning their input and processing across the cluster, a technique called partitioned parallelism. When the instances of a partitioned operator need to communicate to non-local instances of the next operator in the pipeline, the communication occurs through the Exchange[13] operator. We describe the Exchange design and the necessary extensions to it for implementing a partitioned parallel CQ dataflow consisting of sequence-preserving operators that *require* their input to be in arrival order.

In this configuration, a scheme that naively applies the previous technique for replicated processing without accounting for the cross-machine communication within a parallel dataflow quickly becomes unreliable. We show that this technique, called cluster pairs, leads to a mean-time-to-failure (MTTF) that falls off quadratically with the number of machines. Moreover, the dataflow stalls during recovery reducing availability. Instead, embedding the logic for replication and recovery within the Exchange can improve MTTF so that it falls off linearly with the number of machines. In Section 5, we describe the Flux design which achieves this MTTF.

4.1 Exchange

In a parallel database, an Exchange [13] operator is used to connect a stage in a pipeline where the producer's output needs to be repartitioned for the consuming operator. For example, in our click stream processing case, a viable way to parallelize the workload is to partition the two group-bys based on `(sess-id, user-id)` at the first level, and `(user-id)` at the second. In this case, the output

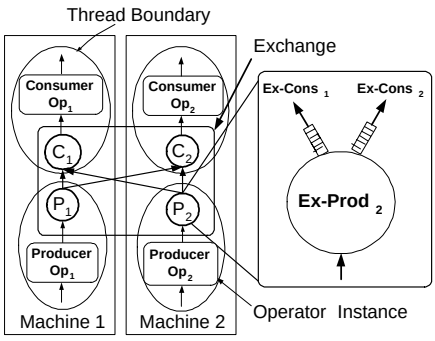

Figure 6: ***Exchange Architecture***

from the first group-by must be repartitioned on the `(user-id)` attribute and sent to the appropriate instance of the second group-by. The Exchange ensures proper routing of data between the partitioned instances of producer and consumer operators. Typically, the operators are fully declustered, so the number of partitions are equal to the number of machines in the cluster.

Although Exchange is conceptualized as a single operator, it is actually composed of two intermediate operators, Ex-Cons and Ex-Prod (see Figure 6). Typically, there is all-to-all communication at an Exchange stage, so each Ex-Prod instance is connected to all the Ex-Cons instances. An Ex-Cons instance supports the traditional `getNext()` interface, and when invoked simply polls its inputs in a round-robin fashion and returns the incoming data to its consumer operator instance. An Ex-Prod instance connects each producer instance to all consumer instances via Ex-Cons. Ex-Prod encapsulates the routing logic. It calls `getNext()` on the producer instance below and routes the output tuples to the appropriate consumer instance based on the tuple's content and the partitioning of the consumer operator. In our example, if the partitioning was hash-based, Ex-Prod would compute the hash on `user-id` from the lower group-by's output. For other operators, round-robin is another possible partitioning scheme.

In an iterator architecture [14], there is typically (but not necessarily) a thread boundary at the Exchange, so the producer and consumer operator instances are scheduled independently, in separate threads of control, i.e. dataflow segments (see Figure 6). The benefits of Exchange are twofold. First, the operator writer can write relational operators while being agnostic as to whether they will be used in a parallel or single-site setting. Second, since a producer instance is generating output for many consumer instances and vice-versa, placing the producer and consumer instances in different threads allows a certain degree of overlapped execution among the instances.

4.1.1 Input Order and Exchange

There are two modifications needed to make Exchange applicable to CQ operators. The first change is simple: to fit within the Fjord model, the `getNext()` interfaces need to become non-blocking to support scheduling Ex-Prod and Ex-Cons in arbitrary DAGs[22]; otherwise, the dataflow may deadlock. Second, the Ex-Cons needs to be order-preserving. Some CQ operators, like the a sliding window group-by whose window slides with every new input, require that the input arrives in sequential order. In a single site dataflow, if the source provides data in sequential order, then the input to the group-by will also arrive in sequential order. However, once the source stream is partitioned across several machines, for any particular group-by instance, the input will arrive in some arbitrarily interleaved order at its Ex-Cons instance. This interleaving occurs because there are multiple producers feeding the Ex-Cons. The streams output by the individual Ex-Prod instances will however, be in arrival order. Thus, the Ex-Cons instance can recover the arrival order by merging the data from its inputs using their input sequence number (ISN) provided by the source or assigned at arrival time from the ingress operator. For CQ operators that relax this input ordering constraint, Ex-Cons can also be modified appropriately to permit more relaxed ordering.

Another issue that arises is the task of maintaining sequence numbers as tuples are processed through the dataflow. We cannot simply generate new output sequence numbers OSNs at Ex-Prod otherwise the ordering across Ex-Prod instances would be lost. Instead we need to keep the original ISN intact to reconstruct the order at the consumer side. For operators that simply filter the input or perform a one-to-one transformation, the operators just need to ensure that the ISN assigned at the source remains intact. For operators that perform a one-to-many transformation like joins, the ISN for each output tuple is a compound key. The compound key consists of the original ISN, and another that uniquely identifies and orders the tuples generated from that input tuple. For example, for a symmetric join that has two input streams arriving in order, a compound ISN that is the concatenation of the ISNs in the input streams will suffice. For many-to-one transformations like windowed aggregates, the largest ISN in the input that produced the output will suffice. It the task of operator developers to generate correct ISNs.

4.2 Naive Solution: Cluster Pairs

In this section we describe how to make a parallel dataflow highly-available and fault-tolerant in a straightforward manner using the technique in Section 3. Assume we still have a single ingress and egress operator. Also assume we use partitioned parallelism for the entire dataflow. Each machine in the cluster is executing a single-site dataflow except the operators only process a partition of the input and repartition midway if needed.

A naive scheme for parallel fault-tolerance would be to apply the ideas of the previous section only to the operator partitions on each machine that communicate with ingress and egress. Using this technique, we can devote half of the machines in a cluster for the primary instances of the dataflow partitions and the other half for the secondary, with each machine having an associated pair. We call this scheme cluster pairs. We refer to the set of machines running either the primary instances or the secondary as a replica set. With cluster pairs, if a machine fails, the state of operators on the faulty machine is unknown since the machine may contain operators that communicate via Exchange. Thus, a single machine failure in either the primary or secondary machines renders the entire dataflow replica set useless.

To see this, from our example, imagine a machine in the primary replica set failed with an instance of the two group-by operators on it. Lets call the lower group-by G_1 and the upper group-by G_2. At the time of failure, the secondary partition of G_1 may have already produced, say a hundred tuples, that the primary was just about to send. Now, if we recovered the state from the remaining secondary, all the primary operator instances of G_2 that relied on the failed partition will never receive those hundred tuples. Without accounting for the communication at the Exchange points, to ensure correctness, we must recover the state of the entire parallel dataflow across all the machines in the primary replica set.

The MTTF for this naive technique is the same as the process-pair approach $(\text{MTTF}_d)^2/\text{MTTR}_d$ where MTTF_d and MTTR_d are for a replica set. Since the replica set is partitioned over $N/2$ machines, $\text{MTTF}_d = 2\text{MTTF}_s/N$. Thus, the overall MTTF falls off quadratically with the number of machines:

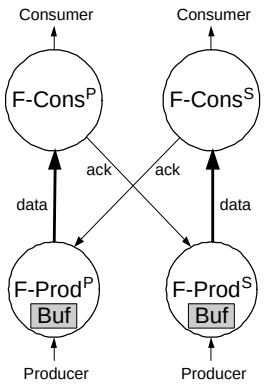

Figure 7: ***Flux design and normal case protocol.***

$\text{MTTF} = 4(\text{MTTF}_s)^2/(N^2\text{MTTR}_d)$.

The catch-up phase is the bulk of our recovery time, it is the determining factor in this equation. If we catch-up all machines in parallel, $\text{MTTR}_d = \text{MTTR}_s$, which improves the system's reliability, but its availability diminishes because the entire dataflow stops processing input during the catch-up phase. If, somehow, we catch-up one machine at-a-time, the availability of the system increases but reliability drops because now $\text{MTTR}_d = \text{N}/2 \times \text{MTTR}_s$. Also, for both of these techniques, we need all $N/2$ machines available before catch-up begins.

Clearly, the cluster pairs technique does not scale well with the number of machines, nor does it provide the high-availability we want during recovery for our critical, high-throughput dataflow applications. If we had a technique that allowed us to properly synchronize the operators at the Exchange points and required recovering only the state of the failed machine, then we could attain much better reliability. The MTTF for such a scheme would be the time for any paired nodes to fail. Since there are $N/2$ paired nodes, and the MTTF for any pair is $(\text{MTTF}_s)^2/\text{MTTR}_s$, overall MTTF is $2(\text{MTTF}_s)^2/(N\text{MTTR}_s)$.

To achieve this improved MTTF, we must modify the Exchange to coordinate operator partition replicas and properly synchronize them after failures and recovery.

5. PARTITION PAIRS

Our analysis in the previous section shows that without proper coordination of operator partitions at communication points, recovery can be inefficient, leading to reduced availability and/or reduced reliability. To ensure correctness for replicated dataflow processing, we must maintain the loss-free and dup-free invariants at their communication points. With cluster pairs, we had two independent dataflows, which were duals of one another, and these properties were maintained through buffering and acknowledgments only at the ingress and egress interfaces. Thus, during catch-up, we were forced to copy the state of the entire replica set. In this section, we build on the protocols for the single-site case and show how to coordinate operator partitions through modification of the Exchange. This design will permit us to recover the dataflow piecemeal and allow the processing for the unperturbed parts of the dataflow to continue, thereby improving both reliability and availability.

Our new operator, Flux , has the same architecture as the Exchange; its constituent operators are called F-Cons and F-Prod . For each F-Cons instance in the primary dataflow, there is a corresponding F-Cons instance in the secondary dataflow, and likewise for the F-Prod instances. We call any such pair of instances *partition pairs*. The F-Cons protocol is similar to egress' during normal processing and take-over, and similar to S-Cons during catch-up. F-Prod 's protocol is similar to S-Prod during normal case and recovery. During normal processing, each F-Cons instance acks received input from one of its F-Prod instances to its dual F-Prod in the replicated dataflow (see Figure 7). We assume Flux uses the order-preserving variant of Exchange to ensure that the input is consumed in the same order for both partition replicas. F-Prod is responsible for routing output to its primary consumers and incorporating acknowledgements from the secondary consumers into its buffer.

There are few salient differences between the Flux protocol and cluster pairs and dataflow pairs schemes. First, the Flux normal case protocol is symmetric. This property makes Flux easier to implement and test because it reduces the state space of possible actions and therefore the number of cases to verify. In the rest of this paper, we artificially distinguish between the primary and secondary versions of the F-Cons and F-Prod . From the point of view of an operator, we use the adjective primary to mean within the same dataflow and secondary to mean within the dual dataflow. Second, Flux handles both multiple producers and multiple consumers. In a partitioned parallel dataflow these producers and consumers are partitions of dataflow operators. Finally, the instances of F-Prod or F-Cons (and operators in their corresponding dataflow segments) are free to be placed on any machine in a cluster subject to one constraint: no two replicas of a partition are on the same machine. This flexibility is useful for administrative and load-balancing purposes as external load varies or machines come and go.

In this section, we describe the modifications necessary to the previous normal-case and recovery protocols to accommodate all-to-all communication at a single stage in a partitioned parallel dataflow. Since there may be many such stages in a dataflow, recovery proceeds bottom-up, recovering one level at a time. We have already shown the base cases for the entry and exit points, and now we will show the inductive step at the Exchange points.

5.1 Flux Normal Case

We specify the normal case forwarding, buffering, and acking protocol Flux uses to guard against unexpected failures.

During normal processing, F-Cons behaves the same as egress except that it manages multiple connections for multiple producers. For each tuple received from a connection to a primary F-Prod , it sends an ack of the tuple's sequence number to the corresponding secondary F-Cons , before considering the tuple for any further processing.

Meanwhile, for each destination, an F-Prod instance obeys the same abstract, normal-case specification for producers shown in Figure 3. The actions remain the same, but the state size increases. It maintains one set of state variables for each consumer partition pair. We use a subscript, i, to denote the variable associated with partition i.

Unlike the ingress operator, however, F-Prod only forwards data to the primary partition, $\texttt{conn}_i\texttt{[P]} = \texttt{SEND}$, and only processes acks from the secondary, $\texttt{conn}_i\texttt{[S]} = \texttt{ACK}$. Finally, once a tuple has been produced, sent to the primary, and acknowledged by the secondary, it is evicted from the buffer, $\texttt{del}_i = \{\texttt{PROD, PRIM, SEC}\}$.

Since F-Prod does not remove any tuples until an ack has been received, all in-flight and undelivered tuples from its replica are in its buffers or will eventually be produced. Thus, this scheme ensures no tuples are lost with up to a single failure per partition pair. In the next section, we describe the take-over protocol which allows the dataflow to continue processing after failures and ensures that no tuples will be duplicated to consumer instances.

5.2 Flux Take-Over

Take-over ensures that regardless of the number of machine failures, as long as only one replica of each partition pair fails, the dataflow will continue to process incoming data and deliver results. Since the the normal case protocol is symmetric, there are only four distinct configurations in which a particular F-Prod^P , F-Prod^S , F-Cons^P , F-Cons^S quartet can survive after failures. Without loss of generality, these cases are: F-Cons^S fails, F-Prod^S fails, F-Cons^S and F-Prod^S fail, or F-Cons^P and F-Prod^S and fail. We describe the F-Prod and F-Cons take-over actions to handle these cases.

For F-Cons , the take-over specification is exactly the same as the one for egress, except the `fail` message now specifies exactly which partition, i, and copy of F-Prod failed. If the primary fails, F-Cons sends a `reverse` message to the secondary and begins receiving data from the secondary. Like S-Cons , it also notes in, `p_fail` if its replica failed, because it is responsible for catch-up, as described in the next section.

For F-Prod , take-over is a combination of the ingress actions and S-Prod actions. Like ingress, when it detects a failure for a consumer instance, it marks all unacknowledged sequence numbers in the corresponding buffer for the failed copy using `ack_all(dest, del)`. In this case, in addition to removing entries with all three marks, this method must remove entries with only the `dest` mark. As a result, if the secondary consumer fails, the method will remove all dangling sequence numbers marked by the secondary. At this point, only entries relevant to the remaining consumer remain in the buffer. If the secondary fails, the buffer only contains undelivered tuples for the primary. If the primary fails, the buffer may contain tuples not yet acked by the secondary or dangling sequence numbers whose tuples have yet to be produced. Moreover, F-Prod also adjusts `del` to ignore the failed partition during the processing between take-over and catch-up.

Like S-Prod , F-Prod notes if its own replica failed and handles `reverse` messages from secondary consumer instances. The action enabled in this case is different (replaces Figure 4-(2)).

`reverse(i,SEC)`		`conn`$_i$`[SEC]:=SEND; r_done`$_i$ `= true;` `del`$_i$`:=del`$_i$`+SEC;`

This state change allows F-Prod to properly forward to both consumers instances if only its replica failed, or to just the secondary if the primary consumer partition i also fails.

In summary, there are four possible cases in which an F-Prod and F-Cons partition may survive. In each of these cases, take-over ensures that all remaining instances continue processing. Moreover, it also maintains the loss-free and dup-free invariants. A case by case analysis shows these facts.

F-Cons^S fails. In this case, F-Prod^P stops processing acks from the faulty F-Cons^S and discards any acks received in the buffer that are not yet generated by the producer. F-Cons^P continues to send acks to F-Prod^S . F-Prod^S continues to process these acks and pull output from the operators below, but no longer forwards data. It is necessary to continue to pull output from operators below to keep both dataflows processing. Otherwise, if the internal buffers are fixed size, they will fill causing back-pressure below and eventually stall the dataflow. For both F-Prod^P and F-Prod^S , the buffer only considers the produced and F-Cons^P marks to determine when to purge a tuple from the buffer. Note, no tuples to F-Cons^P are duplicated since forward routing of data has not changed. F-Prod^P is forwarding tuples to F-Cons^P before and after the failure.

F-Prod^S fails. In this case, processing must continue for the remaining portion of the dataflow downstream of F-Cons^S . Otherwise, it will get further and further out of sync if a standby for F-Prod^S is not available. Since we consider long-running dataflows, F-Prod^P will either have to stall or drop data intended for F-Cons^S , neither of which is desirable.

Thus, F-Prod^P must start feeding F-Cons^S new data, which begins when a `reverse` is received by F-Prod^P . Once a `reverse` is received, all acks from F-Cons^S have been processed. The remaining tuples in the buffer have not been processed by F-Cons^S . The dangling sequence numbers in the buffer will later filter out tuples from operators below that have already been consumed by F-Cons^S . Thus, no data will be duplicated. After the reverse is received, take-over is complete and neither F-Cons sends acks. The two F-Cons are receiving data from the remaining F-Prod , so the dataflow continues making progress.

F-Cons^S and F-Prod^S fail or F-Cons^P and F-Prod^S fail. In both of these cases, since failure messages are ordered and actions atomic, actions for the failure of one of these operators will occur before the other. Since we have shown the protocol works for the first failure, we just need to show that it works for the second.

If either F-Cons fails second, the previous state is one in which the remaining F-Prod is feeding both replicas. After the second failure, it eliminates one of the two connections. In either case, since the broadcast routing was correct before and `ack_all()` leaves only tuples relevant to the remaining connection, forwarding will be correct after. If F-Prod^S fails second, then either F-Cons^P just stops acking and data routing is still correct or the connection to F-Cons^S is reversed. As before, this ensures are acks from F-Cons^S is are received, so the buffer will filter out already acked future upstream. Also, after both failures, there is still a remaining forward connection; hence, the dataflow continues to process.

5.3 Flux Catch-Up

The catch-up phase for a partition pair is similar to one described for a single-site dataflow. In this section, we detail the differences. For the purposes of exposition, we assume no failures occur during catch-up of a single partition. Achieving the idempotency property in this case is akin to that in Section 3.5.

Once the a newly reset node is available or a standby machine is available, the catch-up phase ensues. Each failed dataflow segment is recovered individually, bottom-up. F-Cons initiates catch-up when it recognizes that catch-up for the previous level is complete and that take-over is complete for its F-Prod above (or S-Prod). Like S-Cons , it stalls the operators within its dataflow segment, and transfers state through the StateMover. Once finished, synchronization messages are broadcast to all primary and secondary consumer partitions at the top of the segment and primary and secondary producer partitions at the bottom to fold in the new partition replica.

There are a few differences between catch-up in this case and the single-site case which we outline first and detail next. First, when F-Cons transfers the state of its F-Prod above to a standby, the state for the primary and secondary destinations in its buffers need to be reversed at the new copy. This modification is straightforward: all markings in entries are reversed and cursor positions swapped. Second, F-Cons cannot just stall the incoming connections by setting them to `PAUSE`. Thus, F-Cons pauses the primary incoming connections through a distributed protocol that stalls the outgoing connections from all its primary F-Prods. Finally, the synchronization messages received at F-Prod and F-Cons are handled differently from the ingress and egress operators.

When F-Cons begins catch-up, its primary and secondary producer instances are finished with catch-up. Thus, we are in the first survival scenario; F-Cons^S has failed. If the remaining F-Cons^P just pauses the connection locally and copies over its state (see Figure 5-(2)), F-Prod^P will not properly account for the in-flight data between it and F-Cons^P immediately after the pause. Those tuples no longer exist in F-Prod^P 's buffer because it is not

receiving acks. However, acks for those tuples will arrive after catch-up at F-Prod^P from F-Prod^S via the new F-Cons^S .

These acks will remain in the buffer indefinitely with our current protocol. Hence, we must ensure there is no in-flight data before catch-up begins. To do so, F-Cons^P sends a `pause` message to all of its producers, which upon receipt pause the outgoing connection and enqueue an ack for the pause on the outgoing connection. Once F-Cons^P receives all the incoming pause-acks, it can begin state transfer. A slight complication arises if F-Cons is order preserving. Since a pause-ack can arrive at anytime, it may prevent F-Cons from merging and consuming incoming tuples from other un-stalled incoming connections. This deadlock occurs because when merging in order, inputs from all partitions are necessary to pull in the next tuple in line. Thus, F-Cons must buffer the in-flight tuples internally in order to drain the network and receive all pause acks. Of course, all buffered tuples still need to be acked to F-Prod^S .

Once state movement is finished, both F-Cons^P and the new F-Cons^S broadcast to all producer partitions a `csync` message with the corresponding version numbers. The state changes at F-Prod are now slightly different than ingress to account for the differences in forwarding protocol and the in-flight acks to F-Cons^S (replaces Figure 5-(6)).

`csync(dest,v)`	`(dest=SEC)`	`{ver[dest]:=v;`
		`del:=del+dest;`
		`B.reset(dest);`
		`status[dest]:=ACTIVE;`
		`conn[dest]:=ACK;`

	⋀ `ver[p(dest)]=v;`	`conn[p(dest)]:=` `{SEND};}`
	`(dest=PRIM)`	`{ver[dest]:=v;` `status[dest]:=ACTIVE;`
	⋀ `ver[p(dest)]=v;`	`conn[dest]:={SEND};}`

During state movement, the primary forwarding connection is paused for both F-Prod replicas. Notice the primary connection begins sending only after the sync from the secondary has arrived. This allows F-Prod^S to consume all in-flight acks sent before the movement. Otherwise, F-Prod^S might forward tuples already consumed by F-Cons^P to the new F-Cons^S .

At the top of the data segment, like S-Prod , both F-Prod operators broadcast a `psync` to all remaining F-Cons . Unlike S-Prod , however, F-Prod can have either one or both F-Cons remaining. F-Prod still broadcasts the `psync` but if it is forwarding data to its secondary, it stops immediately and begins processing acks instead, i.e. `conn[SEC]:=ACK`. The F-Cons in the next level also must handle the `psync` messages differently according the following action (replacing Figure 5-(8)).

`psync(dest,v)`		`{ver[dest]:=v;` `status[dest]:=ACTIVE;`
	`dest=SEC` ⋀ `ver[SEC]=v`	`conn[SEC]:=ACK;` `conn[PRIM]:=RECV;}`
	`dest=PRIM` ⋀ `ver[PRIM]=v`	`conn[PRIM]:=RECV;}`
	`else`	`conn[PRIM]:=PAUSE;}`

To avoid missing acks or sending redundant acks, like egress, F-Cons cannot activate a connection upon receiving a `psync` unless the corresponding `psync` from the replica has arrived. Unlike egress, after both `psync`s arrive, F-Cons always sends acks to the F-Prod^S and receives data from F-Prod^P regardless of which was active before catch-up.

5.4 Experiment

In this section, we illustrate the benefits of our design by examining the behavior of the Flux stage in a parallel implementation of our example dataflow. We focus on the throughput of the F-Prod instances during a failure to demonstrate that the dataflow is recovered piecemeal and that it does not stall during recovery. To do so, we implemented a streaming hash-based group-by aggregation operator, our SSFT operators, and Flux within the PostgreSQL open-source code base [1]. In this experiment, we partition this dataflow across four machines in a cluster and place the ingress and egress operators on a separate fifth machine. Each machine has a Pentium III 1.4 GHz CPU, 512MB of RAM and is connected to a 100mpbs switch. We insert a Flux after the first group-by operators to repartition its output on `user-id`.

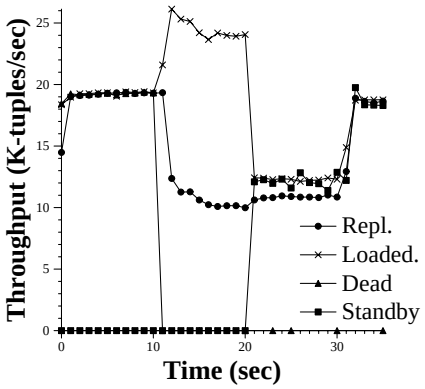

Figure 8: ***F-Prod Throughput During Recovery***

For the purposes of the experiment, to approximate the workload of a high-throughput click stream workload, we have an ingress operator that generates sequentially numbered session-end events as fast as possible. We simulate the first group-by with an operator that stamps each incoming click-stream tuple with a randomly chosen duration, and projects out the `sess-id` field. During catch-up is transfers 100K to its replica to simulate the cost of state transfer. The second group-by is a true hash-based group-by that maintains the maximum and average session duration for each unique `user-id` over all history and outputs results every thousand updates. There are 10000 unique `user-id` values.

At startup, we place a partition of each operator on each of the four machines, numbered 0 to 3, and replicate them using a chained declustering strategy [16]. That is, each primary partition has its replica on the next machine. For example, the primary copy of partition 3 of the first group-by is on machine 3 and its replica is on machine 0. In this configuration, when a single machine fails all four survival scenarios occur in different partitions.

We have not implemented the controller, because it involves well known techniques implemented by standard cluster management software [32]. We introduce a standby machine with operators in their initial state at startup time. We simulate failure by killing the Postgres process on one of those machines, which causes connections to that machine to close and raise an error.

With this setup, Figure 8 shows the total output rate of four F-Prod instances on different machines during a failure (each output tuple is 40 bytes). Three of the F-Prod instance are in the primary dataflow, and the fourth is a standby. In this experiment, we kill a machine at time 10 sec, initiate catch-up of the first group-by at time 20 sec, and initiate catch-up of the second group-by at time 30 sec. Each of these phases exhibits a different behavior, and we describe each in turn. Note, the line labeled dead is for the instance that is killed, standby is for the standby, loaded is for the instance that loses its replica in the secondary data flow, and replicated is for one of the two instances that is unaffected (the other instance is not shown but has the same behavior).

After 10 sec, take-over completes immediately, and the output

rate of the instance which has lost its replica increases. This increase arises because it needs to send data to both primary and secondary consumers. Since there is now one less consumer partition, the output rate of the replicated F-Prod drops (the replica of the dead consumer is delivering results). After catch-up of the first operator, 20 sec, the output rate of the other F-Prod instances approaches the rate of the replicated one. After the catch-up of the second group-by, at 30 sec, the output rates return levels before the failure.

In this experiment, there are two salient feature to note. First, recovery can be accomplished piecemeal as resources become available. Second, even during failure and recovery, the dataflow continues to process and forward data.

6. RELATED WORK

There is a plethora of work on fault-tolerance, availability, and recovery in a variety of fields ranging from theory of distributed computing to practical database recovery. Most related work falls into the category of mechanisms for making generalized computations fault-tolerant. In contrast, our work focuses on a narrower but still generally useful style of computation: CQ dataflows.

The replicated state-machine approach was proposed by Lamport [19] to provide protection against faults in a distributed environment. Schneider [28] provides a survey of the state-machine based approaches. The critical step in this approach is to reach consensus among the replicas for a consistent view of the input sequence. The Paxos [20] algorithm is the most fault-tolerant method for reaching distributed consensus. The process pairs [15] approach is a practical method for implementing the replicated state-machine method for a process. In process pairs, there is a single leader which determines the order of input. If the leader fails, requests may be lost so additional protocols are needed to retry such requests.

There are a number of approaches for checkpointing and rollback recovery schemes for message passing systems. The authors in [11] survey these methods and their tradeoffs. In the Phoenix project [21] they leverage these techniques to provide persistent database client sessions and persistent COM components. These techniques were used originally in the scientific computing community where long-running computations are the norm and reliability is important. Yet, high-availability is not a critical concern for this work. In the CQ scenario, the system needs to be flexible enough to react to incoming data at all times. Checkpointing and stalling the computation during normal processing is unacceptable.

Persistent queues [4] are a messaging abstraction that make messages persistent across failures and can be used for coarse grain load-balancing. For our scenario, they are much too heavyweight because they provide transactional semantics and depend on writes to stable storage.

These previous schemes are black-box techniques that consider input and output to a generic monolithic computation. Their model for requests and responses is client-server rather than a chain of computations, one feeding the next. They do not take advantage of the structure of CQ dataflows – especially parallel dataflows – to provide improved reliability and availability.

The Isis [5] and Horus [6] projects are similar to our work in that they provided a system of programming abstractions for building generic highly-available applications. Isis provides the application programmer reliable communication primitives like group broadcast for ensuring a consistent ordering of broadcast messages for process group, and atomic broadcast for ensuring all or none behavior. Horus is an extensible system in which the programmer can pick, choose, and compose the networking layers necessary for his or her application. While similar in spirit to this work, these primitives provide semantics different than that necessary for partitioned parallel dataflow computation.

There is recent work on the ClustRa system which provides a highly-available and incrementally scalable main memory database for telephony applications [17]. This is a database system with transactional semantics; it performs a workload similar to TPC-C in main-memory for high-availability. In contrast, we focus on a single, long-running dataflow.

7. CONCLUSION

In this paper, we show how to achieve high-availability and fault-tolerance for long-running, high-throughput, parallel dataflows. Our main contribution is a technique for coordinating replicas of operator partitions within a larger parallel dataflow. It is a delicate combination of partitioned parallelism and process pairs. Our technique is more reliable and more available than the cluster pairs approach, which is the straightfoward solution. Our scheme provides online recovery without stalling the ongoing dataflow computation because it allows for recovering the dataflow in piecemeal. The protocols we describe are in encapsulated in an opaque dataflow operator called Flux . Thus, an application developer reuse it in conjunction with a variety of operators to make existing, brittle dataflows more robust.

Given the Flux mechanism, we speculate on some future work. To achieve the same redundancy after a failure, our recovery protocol require some spare machines to be available on which a new secondary copy can be instantiated. If a spare is not available, human intervention is required to introduce one, which leads to an increase in recovery time and possibilities for additional failures. On a cluster-based platform, we should be able to use the other running nodes to create a secondary copy of the failed machine. However, doing so would create imbalances in the ongoing dataflow causing it to under-perform. Integrating load-balancing into this fault-tolerance mechanism is necessary next step.

8. REFERENCES

[1] Postgresql. *http://www.postgresql.org*.

[2] T. Anderson, D. Culler, and D. Patterson. A Case for Networks of Workstations: NOW. *IEEE Micro*, Feb. 1995.

[3] J. Baulier, S. Blott, H. Korth, and A. Silberschatz. A database system for real-time event aggregation in telecommunication. *VLDB*, 1998.

[4] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *ACM SIGMOD*, pages 112–122, 1990.

[5] K. Birman et al. Isis: A system for fault-tolerant distributed computing. Technical report, Cornell, 1986.

[6] K. Birman et al. The horus and ensemble projects: Accomplishments and limitations. Technical report, Cornell, 1997.

[7] D. Carnery et al. Monitoring Streams - A New Class of Data Management Applications. In *VLDB*, 2002.

[8] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *CIDR*, 2003.

[9] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

[10] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, June 1992.

[11] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical report, CMU, 1997.

[12] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services, 2002.

[13] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*, 1990.

[14] G. Graefe. Query Evaluation Techniques for Large

Databases. In *ACM Computing Surveys*, 2002.

[15] J. Gray and A. Reuter. *Transaction Processing – Concepts and Techniques*. Kaufmann, 1993.

[16] H. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. *ICDE*, 1990.

[17] S. Hvasshovd et al. The ClustRa Telecom Database. *VLDB*, 1995.

[18] C. Kruegel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful Intrusion Detection for High-Speed Netrowks. *IEEE Symposium on Security and Privacy*, May 2002.

[19] L. Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 1978.

[20] B. Lampson. The ABCD's of Paxos. *PODC*, Aug. 2001.

[21] D. Lomet and R. Barga. Phoenix Project: Fault Tolerant Applications. *SIGMOD Record*, June 2002.

[22] S. Madden and M. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *ICDE*, February 2002.

[23] M. Mehta and D. DeWitt. Managing Intra-operator Parallelism in Parallel Database Systems. In *VLDB*, 1995.

[24] R. Motwani et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. *CIDR*, 2003.

[25] V. Paxon. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 1999.

[26] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

[27] F. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *Transactions on Computer Systems*, May 1984.

[28] F. Schneider. Implementing Fault-Tolerant Services Using the State-Machine Approach: A Tutorial. *Computing Surveys*, Dec. 1990.

[29] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering*, Mar. 1986.

[30] T. Urhan and M. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, pages 27–33, 2000 2000.

[31] G. Vigna, W. Robertson, V. Kher, and R. A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. *ACSAC*, 2003.

[32] W. Vogels et al. The Design and Architecture of the Microsoft Cluster Service. *FTCS*, 1998.